

REMARKS

By the present amendment, Applicants have amended Claims 6, 12, 14, and 18. Claims 1-20 remain pending in the present application. Claims 1, 12, and 18 are independent claims.

Applicants' representative called the Examiner on October 4, October 7, and October 11, 2005, leaving voice mail messages requesting the Examiner to contact Applicants' representative to set up an interview. Applicants' representative contacted the Examiner's former supervisor, Richard Chilcot, on October 11, who advised that the Examiner's current supervisor is Alexander Kalinowski. Applicants' representative left a voice mail with Examiner Kalinowski on October 11, also advising that an interview was desired. Applicants' representative has received a message from the Examiner on October 13, 2005 setting up an interview for October 19 at 1:00 p.m..

Applicants respectfully request that the Examiner withhold issuing an official Office Action in response to the present amendment until after the interview scheduled for October 19, 2005.

In the recent Office Action the Examiner rejected Claims 6, 14-17, and 18-20 under 35 U.S.C. § 112, second paragraph, as being indefinite. Applicants have amended Claim 6 to change the dependency from Claim 3 to Claim 2 and to add the Internet customer's web browser as an element to the claimed combination. Applicants have amended Claim 14 in lines 3 and 4 to clarify that communications from the Web server (the second computer) to the first computer are by electronic mail over the Public Internet. Claim 18 has been amended to change "querying" to "linking" to pages on the website. In addition, Claim 12 has been amended at line

BEST AVAILABLE COPY

34 to change “merchant computer” to “first computer” in order to correct an antecedent basis problem. No new matter has been added by the foregoing amendments. Applicant respectfully submits that Claims 6, 12, 14-17 and 18-20, as amended, meet the specific requirements of 35 U.S.C. § 112, second paragraph.

In the recent Office Action the Examiner rejected Claims 1-7 under 35 U.S.C. § 103(a) as being unpatentable over Atrex (archived materials from atrex.com, published February 2001) in view of iCat (archived materials from iCat.com, published January 1997). This rejection is respectfully traversed.

Three criteria must be met to establish a *prima facie* case of obviousness. The third criterion is that the prior art reference (or combination of references) must teach or suggest all the claims limitations. MPEP 2143. Consequently, if at least one claim limitation is not taught or suggested by the references, the rejection under 35 U.S.C. § 103(a) is not proper.

With respect to independent Claim 1, the computer program means includes “fifth code means for updating said inventory database to reflect sales made on the website and at the point-of sale, the fifth code means being processed on the first computer without using any server side software, installation, or setup for processing of any computer instructions on the at least one second computer other than Internet and Web protocols.” The Examiner refers to the C-Ecommerce Link Info Page of the Atrex reference to show means for maintaining and updating the inventory database to reflect sales made on both the website and at the point-of-sale. The Examiner states that Atrex is database software that includes the traditional means for maintaining and updating the database for point-of-sale transactions, and the partnership

with TrustCart added the functionality to maintain and update the database for sales made on the website.

Applicants point out that the claim limitation requires that there be no server side software, other than means for transmitting documents by HTTP, SMTP, or other Internet protocols. The reference forwarded by the Examiner includes the TrustCart home page, identified by the URL web.archive.org/web/20010331105119/www.trustcart.com/, which clearly states that TrustCart “comes with your own secure server.” That is, the customer posts his order to the TrustCart website by HTTP, the TrustCart website then processing the order for downloading by the merchant. This is made explicit by the Atrex FAQ published at the URL www.1000years.com/ecommerce.shtml (a copy of which is attached hereto as Appendix A), which explains that Millennium Software provided a utility to extract inventory information from Atrex for upload to the TrustCart server, and for retrieving online orders from TrustCart and inserting the orders directly into the Sales Order function in Atrex.

The Atrex/TrustCart system and software is fundamentally different from the system described in Claim 1, and in the specification. The system of Claim 1 has no server side software, which is explicitly noted as a limitation in the fifth code means of Claim 1. The system provides for publishing HTML information pages on a web hosting provider service, the HTML pages having script embedded therein for creating a client-side shopping cart on the customer's web browser. Such a technique was well known at the time of the present invention. For example, “How to Set Up and Maintain a Web Site”, 2nd edition, by Lincoln D. Stein, published in 1997, shows how to set up a client-side shopping cart using JavaScript at pp. 627-641 (a copy of which is attached hereto as Appendix B). However, as noted at p. 636 of the Stein reference,

the state of the art at that time provided that when the customer clicks the submit button, the form is returned to the server for processing by a server side Common Gateway Interface (CGI) script.

By contrast, the client side script provided by the present invention provides that, according to the method of payment selected by the customer, the order form is sent to the first (or merchant) computer by e-mail, or the client is directed to a third party SSL or other secure server (separate from the web hosting service that published the inventory pages), which then e-mails the order to the first (or merchant) computer (or possibly posts it to the web hosting computer for later download). The e-mail may be sent by having the script invoke the customer's web browser e-mail capability (well known at the time for e-mailing questions or comments to a webmaster) using an e-mail address provided in the script or an HTML anchor and pasting the order into the body of the e-mail message. The first computer is provided with software to parse the e-mail (similar to a CGI script that could be used on a web server) and thereby present orders and update the inventory in the database. The Atrex and TrustCart references make no mention of e-mailing the orders from the customer directly to the e-mail account of the computer running the Atrex software without intermediate processing by the TrustCart computer. To the contrary, the TrustCart web pages and Atrex FAQ make it clear that the order form is sent from the customer to the TrustCart server for processing by the "secure server" provided by TrustCart.

There is server software in the Atrex/TrustCart system. Consequently, the fifth code means limitation of "without using any server side software" of Claim 1 is not met by the Atrex/TrustCart references. Therefore, Applicants respectfully submit that independent Claim

1 is not unpatentable over Atrex in view of TrustCart, and independent Claim 1 and corresponding dependent Claims 2-11 are allowable over the prior art of record.

Regarding independent Claim 12, the claim contains the limitation of "sixth computer readable program code for processing e-commerce transactions, the sixth program code being processed on the first computer without any server side software, installation, or setup for processing of any computer instructions on the at least one second computer other than Internet and Web protocols, resulting in the data being in the same computer readable format as the point-of-sale transactions." The Examiner applied the same combination of Atrex in view of iCat to reject Claim 12 under 35 U.S.C. § 103(a) for the reasons set forth above in rejecting Claim 1. Applicants traverse the rejection for the same reasons set forth in the traverse of Claim 1. Therefore, Applicants respectfully submit that independent Claim 12 is not unpatentable over Atrex in view of TrustCart, and independent Claim 12 and corresponding dependent Claims 13-17 are allowable over the prior art of record.

With respect to independent Claim 18, the Examiner applied the same combination of Atrex in view of iCat to reject Claim 18 under 35 U.S.C. § 103(a) for the reasons set forth above in rejecting Claim 1. Applicant has further amended Claim 18 at lines 5-9 to provide a further limitation of "the website including a home page, a plurality of pages describing items of inventory offered for sale via both point-of-sale and the website, index pages, and client-side script embedded in the pages operable on a client computer viewing the pages for ordering the items of inventory via e-mail or a third-party secured transaction server." For the reasons cited in the traverse of Claim 1, Applicants respectfully submit that independent

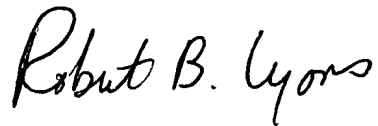
Serial No. 09/902,797
Art Unit: 3627

Attorney Docket No. 19255.04
Confirmation No. 7452

Claim 18 is not unpatentable over Atrex in view of TrustCart, and independent Claim 18 and corresponding dependent Claims 19-20 are allowable over the prior art of record.

For the foregoing reasons, Applicant respectfully submits that the present application is in condition for allowance. If such is not the case, the Examiner is requested to kindly contact the undersigned in an effort to satisfactorily conclude the prosecution of this application.

Respectfully submitted,

A handwritten signature in black ink that reads "Robert B. Lyons". The signature is written in a cursive, flowing style.

Robert B. Lyons
Registration No. 40,708
(703) 486-1000

RBL:rbf
Attachments

Features

Awards

Screens

Resellers

E-Commerce

Online shopping is quickly becoming a major way of doing business. With this in mind, Millennium Software has partnered with two online shopping solutions: TrustCart and Account Wizard.

Account Wizard provides for online shopping as well as some enhanced account history retrieval, that allows your repeat clients to view their outstanding orders and prior purchase history via their web browser. Account Wizard includes an external utility to send Atrex information to the Account Wizard system, automatically, on a routine scheduled basis. An additional utility has been developed, by Millennium Software, to allow you to retrieve online orders from the Account Wizard system and insert the orders directly into the Sales Order function in Atrex. You can visit the Account Wizard website at <http://www.accountwzard.com> for additional information or <http://www.americanphonesystems.com/> to see a sample site using the Account Wizard system.

The TrustCart system provides for simple online shopping as well as web hosting. Millennium Software has developed a small utility to allow you to quickly and easily extract inventory information and send it up to the TrustCart system. This same utility can also be used for retrieving online orders from TrustCart and inserting the orders directly into the Sales Order function in Atrex. You can visit the TrustCart website at <http://www.trustcart.com> for additional information or <http://www.chippewa-bird.com/> to see a sample site using the TrustCart system.

Both utilities, developed by Millennium Software to interface with the above e-commerce sites, are available on our downloads page.



STO NUOVO PROCESSORE SCALDA
MEGLIO DI UN FORNO A LEGNA!

LinkExchange

☒ TradeBanners Member

Millions of TradeBanners Served!

<http://www.1000years.com>
<http://www.atrex.com>

Last Modified: June 1, 2005

HOW TO SET UP AND MAINTAIN A

WEB SITE

Lincoln D. Stein



ADDISON-WESLEY

An imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts • Harlow, England • Menlo Park, California
Berkeley, California • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

APPENDIX B (p. 1)

HOW TO SET UP AND MAINTAIN A WEB SITE

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book and Addison Wesley Longman, Inc. was aware of a trademark claim, the designations have been printed with initial capital letters.

The publisher offers discounts on this book when ordered in quantity for special sales.

The authors and publishers have taken care in preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For more information, please contact:

Corporate & Professional Publishing Group
Addison Wesley Longman, Inc.
One Jacob Way
Reading, Massachusetts 01867

Library of Congress Cataloging-in-Publication Data

Stein, Lincoln D., 1960-

How to set up and maintain a Web site/Lincoln D. Stein.

-Second Edition. p. cm.

Includes index.

ISBN 0-201-63462-7 (pbk. : alk. paper)

1. World Wide Web (Information retrieval system) 2. Web sites-Design. 3. World Wide Web servers. I. Title.

TK5105.888.S74 1997

005.2'76--dc21

96-46116

CIP

Copyright © 1997 by Addison Wesley Longman, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada.

Text printed on recycled and acid-free paper.

ISBN 0201634627

8 9 1011213 MA 03 02 01 00

8th Printing February 2000

APPENDIX B (p. 2)

the remaining elements in the form required text fields. If they were not, each of the required fields would have to be checked individually by name.

If the form passes all the checks, `validateForm()` returns `true`. In this example, the form's submission ACTION is set to the `printenv.pl` program (Chapter 9) just so we can see it do something. In real life it would point to a CGI script that processes the request in some way.

JavaScript Bugs and Security Holes

The initial release of JavaScript in Netscape 2.0 was a bit rocky. Within a few days of its release in March 1996, Internet programmers had discovered a series of security holes in the language that ranged from the trivial to the dangerous.

Some problems were minor annoyances. For example, it was possible to write a JavaScript program that would make the browser send out e-mail without the user's knowledge or permission. Since the user's return address is attached to outgoing mail, this allowed the operator of a Web site to retrieve the e-mail address of everyone accessing his pages.

Other problems were more serious. For example, the original release of JavaScript could scan a user's hard disk and upload directory listings to a remote server somewhere on the Internet. It was also possible to trick the user into uploading the contents of a private file by hiding the file upload function within an innocuous button. A JavaScript could stay resident long after its page had closed, monitor all pages the user viewed, and submit a report on the user's browsing habits to a remote server.

At the time this was written, Netscape 2.01 had addressed several of these problems, and Netscape promised to fix the remaining ones in version 3.0. If you are using an older version of Netscape Navigator, you should upgrade and advise your users to do so as well.

A Shopping Cart

Shopping cart pages allow the user to browse through online catalogs and "collect" items throughout the session. Whenever the user feels the urge to make a purchase, she presses an "Order" button and the current item is added to the user's "shopping cart." When the user's done browsing, she has the option of placing an order for the items in her cart.

Shopping cart applications have traditionally been written as CGI applications. Shopping carts are tricky to write in CGI because of the need to keep track of the user's browsing history over a long series of individual CGI transactions. Usually these applications require that a file or database record be written to disk on the server's side of the connection in order to

APPENDIX B (p.3)

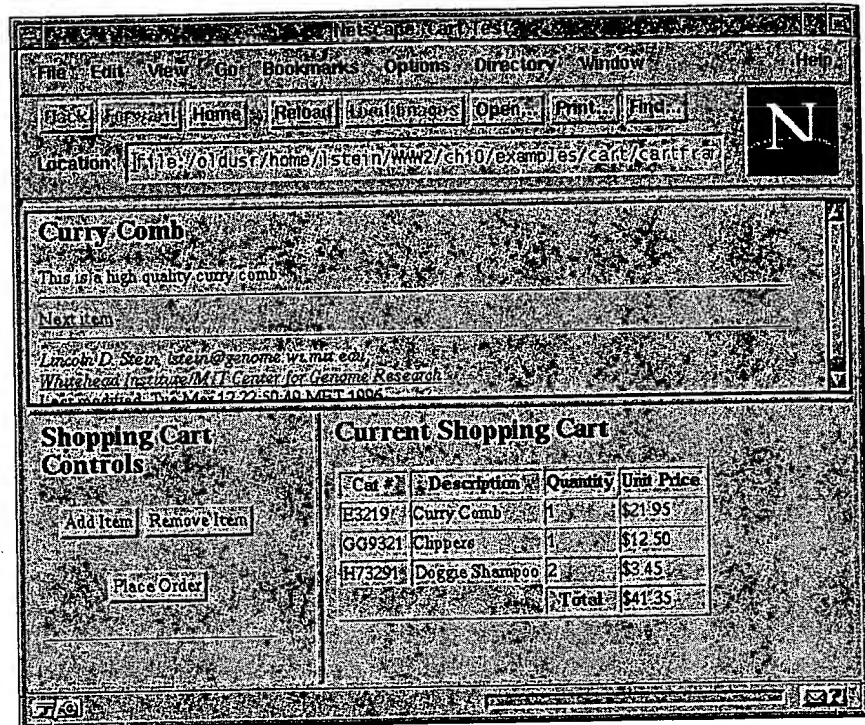


FIGURE 10.11 A JavaScript Shopping Cart

keep track of the user's requests, and that mechanisms be designed to delete the old files after some period of time has elapsed.

Although a large number of CGI shopping cart scripts have been written, they tend to be specific for the particular site where they were designed and are not easily transportable.

You can write a basic shopping cart script easily in JavaScript, however, by taking advantage of frames and global variables. The script runs in one frame, while the user views the catalog pages in a separate catalog frame. To order a displayed item, the user clicks on an "Add Item" button. The script reads the catalog frame document to get the item information and then adds it to a global variable that keeps track of the contents of the cart. The total cost of the items in the cart variable is continuously developed in a third frame (Figure 10.11). To delete an item, the user can press the "Remove Item" button.

When the user has finished shopping, she presses a "Place Order" button, and the script pops up a new window that displays an order form (Figure 10.12). The order form allows the user to enter her name and billing information as well as giving her a chance to delete items or change the ordered quantities. When the user is satisfied, she presses "Place Order" again and the order is sent off to a CGI script for processing.

APPENDIX B (p.4)

Although the example shopping cart script shown here isn't complete, it's a good example of how to use JavaScript's object-oriented properties, and it's at least functional enough to get you started on a real application.

The shopping cart system requires several files. The main entry point to the catalog is a frameset named *cart-frame.html* shown here:

```
<HTML> <HEAD>
<TITLE>Shopping Cart</TITLE>
</HEAD>
<FRAMESET ROWS="60%,*">
  <FRAME SRC="item1.html" NAME="catalog">
  <FRAMESET COLS="30%,*">
    <FRAME SRC="control_panel.html" NAME="control_panel">
    <FRAME SRC="blank.html" NAME="cart">
  </FRAMESET>
</FRAMESET>
<HTML>
```

This frameset creates three panels. There's a large central panel at the top of the window that contains catalog entries named "Catalog." On the lower left is a small panel named "control_panel." It contains the controls for the cart and is the place that does most of the JavaScript work. The lower right panel, "cart," is reserved for a display of the shopping cart's current contents. Since it will be rewritten as soon as the control panel's script begins to execute, it doesn't much matter what URL we initially specify when we create this panel. We initialize it by using an empty HTML file named *blank.html*.

Let's consider the contents of the catalog pages first. The catalog pages can contain any arbitrary HTML: links, in-line images, navigation bars, and even JavaScripts are legal. The only limitation is that pages that contain something orderable must contain some information about the item formatted in a way that the control panel script can pick up. Here's an excerpt from a short catalog page:

```
<H1>Curry Comb</H1>
<FORM>
<INPUT TYPE="hidden" NAME="description" value="E3219:Curry
Comb:21.95">
</FORM>
This is a high-quality curry comb, created of the finest
materials and lovingly handcrafted by our gifted staff for
many years of service.
<HR>
<A HREF="item2.html">Next item</A>
<HR>
```

The important thing to notice is that the page contains a form, which in turn contains a single hidden field named "description." This field, which is not displayed to the user, has a value consisting of the item's catalog number,

APPENDIX B (p. 5)

Please confirm this order list. Change the quantity ordered to 0 to cancel an order, to submit the order.

Cat #	Description	Quantity
E3219	Curry Comb	1
GG9321	Clippers	1
H73291	Doggie Shampoo	2

Your name:

Customer Number:

PO Number:

Shipping address:

FIGURE 10.12 The Shopping Cart's Order Form

its name, and price, all separated by the ":" character. In this case, the value "E3219:CurryComb:21.95" is to be interpreted as catalog number E3219, description "Curry Comb," price \$21.95. A hidden field of this sort is all that's needed to make it possible to add this item to the shopping cart.

All the difficult work is done in *control_panel.html*, whose complete code is shown in Figure 10.13.

```

01 <HTML> <head>
02 <TITLE>Shopping Cart Controls</TITLE>
03 <SCRIPT>
04 <!-- hide the script from other browsers
05
06 // Sorting function. Given an object containing
07 // properties, creates an array (1 based!) with the

```

APPENDIX B (p. 6)


```
08 // properties sorted in alphabetic order.
09 function sortKeys (object) {
10     this.length=0;
11     for (var a in object) {
12         var pos = 1;
13         while (pos <= this.length) {
14             if (this[pos] > a)
15                 break;
16             pos++;
17         }
18         for (var i=this.length;i >= pos; i--)
19             this[i+1]=this[i];
20         // Put us where we belong
21         this[pos]=a;
22         this.length++;
23     }
24     return this;
25 }
26
27 // Split the string catalog:description:price into
28 // its three component pieces
29 function catEntry (string) {
30     var firstColon = string.indexOf(":");
31     var lastColon = string.lastIndexOf(":");
32     this.catNo = string.substring(0,firstColon);
33     this.description = string.substring(firstColon+1,lastColon);
34     this.price = string.substring(lastColon+1,string.length);
35     return this;
36 }
37
38 // Add an item to the bag.
39 function add(item) {
40     if (item == null)
41         return;
42     if (this.cart[item.catNo])
43         this.cart[item.catNo]++;
44     else
45         this.cart[item.catNo]=1;
46     this.entries[item.catNo]=item;
47 }
48
49 // there's no way to actually delete
50 // a property, so we do a copy operation
51 function remove(item) {
52     if (item == null)
53         return;
54     var temp = new array();
55     for (var a in this.cart) {
56         if (a == item.catNo) {
57             if (this.cart[a] > 1)
58                 temp[a]=this.cart[a]-1;
59             } else
60                 temp[a]=this.cart[a];
61         }
62     }
63     this.cart = temp;
```

APPENDIX B (p. 7)

```

63 }
64
65 // Turn a floating point number into a nicely formatted
66 // price with two decimal places
67 function formatAsPrice(price) {
68     var cents = Math.floor((100*price)%100);
69     var dollars = Math.floor(price);
70     if (cents == 0)
71         cents = "00";
72     else if (cents < 10)
73         cents = "0" + cents;
74     return dollars + "." + cents;
75 }
76
77 // List the contents of our cart
78 function list() {
79     var totalPrice = 0.0;
80     var result = "<table border>"
81     result += "<th>Cat #<th>Description<th>Quantity<th>Unit Price";
82     var keys = new sortKeys(this.cart);
83
84     for ( i = 1; i <= keys.length; i++ ) {
85         var a = keys[i];
86         var catNo = this.entries[a].catNo;
87         var description = this.entries[a].description;
88
89         result += "<tr><td>" + catNo +
90             "<td>" + description +
91             "<td>" + this.cart[a] +
92             "<td>$" + this.entries[a].price;
93
94         totalPrice += this.entries[a].price * this.cart[a];
95     }
96     result += "<tr><td><td><th>Total<td>$" + formatAsPrice(totalPrice);
97     result += "</table>";
98     return result;
99 }
100
101 // Create an order form from the list
102 function make_orderForm() {
103     var result = "Please confirm this order list. Change the quantity ordered ";
104     result += "to 0 to cancel an item. Press \"Order\" to submit the order.";
105     result += '<form action="/cgi-bin/printinv.pl" method=POST>';
106     result += "<table BORDER>"
107     result += "<th>Cat #<th>Description<th>Quantity";
108     var keys = new sortKeys(this.cart);
109     for ( i = 1; i <= keys.length; i++ ) {
110         var a = keys[i];
111         var catNo = this.entries[a].catNo;
112         var quantity = '<input type="text" name="item:' + catNo +
113             '" value="' + this.cart[a] + '" size=2>';
114         result += "<tr><td>" + catNo +
115             "<td>" + this.entries[a].description +
116             "<td>" + quantity;
117     }

```



```

118     result += "</table><p>";
119     result += '<table><tr><th>Your name<td><input type="text" name="name">';
120     result += '<tr><th>Customer Number<td><input type="text" name="custNo">';
121     result += '<tr><th>PO Number<td><input type="text" name="PO"></table><p>';
122     result += '<strong>Shipping address:</strong><br>';
123     result += '<textarea name="address" rows=4 cols=40></textarea><p>';
124     result += '<input type="submit" value="Place Order">';
125     return result;
126 }
127
128 function show(aDoc) {
129     aDoc.clear();
130     aDoc.open("text/html");
131     aDoc.writeln("<HTML><HEAD><TITLE>Current Shopping Cart</TITLE></HEAD>
132     <BODY>");
133     aDoc.writeln("<H1>Current Shopping Cart</H1>");
134     aDoc.writeln(this.list());
135     aDoc.writeln("</BODY></HTML>");
136     aDoc.close();
137 }
138 // create a new window and display an order form
139 // within it
140 function order() {
141     var orderWin = window.open("");
142     var a = orderWin.document;
143     a.clear();
144     a.open("text/html");
145     a.writeln("<HTML><HEAD><TITLE>Order Form</TITLE></HEAD><BODY>");
146     a.writeln("<H1>Order Form</H1>");
147     a.writeln(this.make_orderForm());
148     a.writeln("</BODY></HTML>");
149     a.close();
150 }
151
152 // blank array with nothing in it.
153 function array() {
154 }
155
156 // Constructor for the cart object
157 function cart() {
158     this.cart=new array();
159     this.entries = new array();
160     this.add=add;
161     this.remove=remove;
162     this.list=list;
163     this.show=show;
164     this.order=order;
165     this.make_orderForm = make_orderForm;
166     return this;
167 }
168
169 // The description of the item, the catno and the price are
170 // found in a hidden field named "description" in the first
171 // form of the current page.

```

APPENDIX B (p. 9)

```

172 function getCurrentItem() {
173     if (parent.catalog.document.forms.length == 0)
174         return null;
175     var itemDesc = parent.catalog.document.forms[0].description.value;
176     if (itemDesc == null)
177         return null;
178     return new catEntry(itemDesc);
179 }
180
181 // GLOBAL INITIALIZATION - CREATE A NEW CART OBJECT
182 theCart = new cart();
183
184 // end hiding -->
185 </SCRIPT>
186 </HEAD>
187
188 <BODY onLoad="theCart.show(parent.cart.document)">
189 <H1>Shopping Cart Controls</H1>
190 <FORM NAME="form1">
191     <CENTER>
192     <INPUT TYPE="button" NAME="add" VALUE="Add Item"
193         onclick="theCart.add(getCurrentItem());
194             theCart.show(parent.cart.document)">
195     <INPUT TYPE="button" NAME="delete" VALUE="Remove Item"
196         onclick="theCart.remove(getCurrentItem());
197             theCart.show(parent.cart.document)">
198     <P>
199     <INPUT TYPE="button" NAME="order" VALUE="Place Order"
200         onclick="theCart.order()">
201     </CENTER>
202 </FORM>
203 <HR>
204 </BODY> </HTML>

```

FIGURE 10.13 Code for Shopping Cart Control Panel

The main data structure used by the control panel code is a "shopping cart" object called "cart." It's actually a completely new class that has the properties and methods necessary to maintain a list of items in the shopping cart and keep track of how many units of each type of item the user wants to order.

The shopping cart is defined by the function `cart()` (lines 156-167) using the object definition syntax described above. A cart has two properties, *cart* and *entries*. The *cart* property keeps track of the number of each item in the cart. If the user wants to order 37 curry combs, for example, its entry in the array would look like:

```
theCart.cart["E3219"] = 37
```

This property takes advantage of the fact that you can use strings as array indexes in JavaScript.

APPENDIX B (p. 10)

The *entries* property is also indexed by catalog number. However, it contains information about the item, such as its description and price, that doesn't change during the session. Although we could store the colon-delimited description information directly in this array, like this

```
theCart.entries["E3219"] = "E3219:Curry Comb:21.9"
```

it's cleaner and more extensible to use yet another type of object called *catEntry* (lines 27–36) to keep track of this information. This object has properties named *catNo*, *description*, and *price*, and can easily be extended to carry more information. A new *catEntry* is created from the colon-delimited string in this way:

```
entry = new catEntry("E3219:Curry Comb:21.9");
```

This is what gets stored in the shopping cart's *entries* array. Individual fields are then accessed like this:

```
theCart.entries["E3219"].price => 21.9
theCart.entries["E3219"].description => "Curry Comb"
theCart.entries["E3219"].catNo => "E3219"
```

Most of the code in the script are definitions for the shopping cart object's methods (Table 10.15):

TABLE 10.15 Shopping Cart Methods

Method	Description
<code>add(item)</code>	Add an item to the cart
<code>remove(item)</code>	Remove an item from the cart
<code>list()</code>	Create a list of the contents
<code>show()</code>	Display the cart contents in a frame
<code>make_orderForm()</code>	Create the order form
<code>order()</code>	Display the order form

The most important methods are `add()`, `remove()`, `show()`, and `order()`. `add()` (lines 38–47) puts a new item into the shopping cart. It expects the item to be in "catalog:description:price" format. If an item of this type is already in the cart, the item count kept in the *cart* array will be bumped up by one; otherwise a new entry is created. Similarly, `remove()` (lines 49–63) removes the indicated item from the cart. If an item of this type isn't found in the cart, nothing happens.

`show()` and `order()` display the contents of the cart. `show()` (lines 128–136) expects a single argument giving it the document to display the cart inside. Whatever is currently in the document is erased and replaced by a table that `show()` creates on the fly. For each item in the cart, the table gives its catalog number, its description, the number of items in the cart, and the unit price for the item. In addition, the table totals up and displays

the cost of the entire purchase. (Issues of sales tax and discount coupons are conveniently ignored in this example!)

`order()` (lines 138–150) does much the same thing as `show()`. In this case, however, `order()` pops up a completely new window and synthesizes an order form. Like `show()`, `order()` creates a table showing each item in the cart. The main difference is that the quantity field is editable (it's part of a form). This allows the user to change the number of items to order, or to cancel a particular item entirely by setting its quantity to zero. In addition to the table, there are the usual fields for the user's name, shipping address, and billing information. (The example requests a PO number—you can replace it with a credit card number if you dare).

When the order form is submitted, its contents are sent off to a CGI script. Things are arranged so that each item's catalog number becomes a separate parameter in the CGI query string. For example, if the user were ordering two curry combs (catalog number E3219) and one clipper (catalog number GG9321), the query string would contain:

```
item:E3219=2&item:GG9321=1&...
```

In this example I just point the order form at a CGI program that echoes back the contents of the order form. Follow the outline of the "user feedback form" in Chapter 9 to arrange for the order to be e-mailed or filed in some way. To recover the list of items, the script should search for all parameters beginning with the text "item:" and recover the catalog number and order quantity.

In addition to the methods for the shopping cart object, the `<SCRIPT>` section contains a few utility functions. One of these, `sortKeys()` (lines 6–25) is an example of how to perform a simple alphabetic insertion sort in JavaScript. Another function, `getCurrentItem()` (lines 169–179), fetches the colon-delimited item description from the current catalog page in this way:

```
var itemDesc =  
    parent.catalog.document.forms[0].description.value;
```

The colon-delimited description is then turned into a *catEntry* object and returned to the caller, who adds it to or removes it from the shopping cart.

Another useful utility function is `formatAsPrice()` (lines 65–75). This function turns a floating-point number into a fixed-point number with two decimal places.

Once the shopping cart object is defined, the rest of the code is straightforward (lines 181–204). At the very end of the `<SCRIPT>` section, we create a global shopping cart object named "theCart." This global object will keep track of all the user's selections. Next we define a single form that contains three buttons, each with its own *onClick* event handler. The buttons named "add" and "delete" fetch the item from the current catalog page by calling `getCurrentItem()`. "add" adds this item to the

APPENDIX B (p. 12)

count coupons

`show()`. In this
 row and synthe-
 showing each
 field is editable
 ber of items to
 quantity to zero.
 er's name, ship-
 ests a PO num-
 re).
 at off to a CGI
 ber becomes a
 f the user were
 e clipper (cata-

l program that
 ine of the "user
 be e-mailed or
 ould search for
 ver the catalog

the `<SCRIPT>`
`s()` (lines 6-25).
 insertion sort in
 179), fetches the
 e in this way:

value;

ntry object and
 shopping cart.
 es 65-75). This
 t number with

of the code is
`<SCRIPT>` section,
 t." This global
 define a single
 event handler.
 om the current
 his item to the

shopping cart by invoking its `add()` method. "delete" does the reverse. Both buttons then call the shopping cart's `show()` method to update the display. Because this method needs to be told which document to write into, we point it at the frame named "cart" using the expression:

```
theCart.show(parent.cart.document);
```

Because we'd like the empty table to be shown when the document first loads, we also call `theCart.show()` in the window's *onLoad* method (defined in the `<BODY>` tag).

The button named "order" just calls the "cart" object's `order()` method: the cart takes care of all the rest.

Making the Shopping Cart "Remember" the User's Purchases

There's one major problem with implementing a shopping cart (or any other state-maintaining page) in JavaScript. The problem is that Netscape *reloads* the script every time the user changes the size of the window or the relative positions of the frames. The unfortunate side effect of this is that the contents of all the scripts cart global variables are wiped clean and the user makes the annoying discovery that her shopping cart has been completely emptied! This also happens if the user temporarily surfs off somewhere else for a while and then comes back to your page, or if the browser crashes before the user submits the order form.

The solution to this is to make the script remember the user's state between accesses using a "magic cookie." We'll create and update a cookie containing the current list of selected items whenever the table of selections is displayed. The cookie will remain valid for one hour from its creation date. If any of the shopping cart pages are reloaded during this time period, the browser will send the cookie back to our script, and we use it to reinitialize the shopping cart. This means that the user can jump to another page somewhere else, browse it for a while, or even quit the browser completely; when she comes back to the shopping cart page, she finds it still fully stocked.

We'll need new methods to create a new cookie from the "cart" object and to reinitialize the cart from an old cookie. A cookie is just a specially formatted string in the form:

```
COOKIE_NAME=COOKIE_VALUE; expires=EXPIRATION_DATE
```

`COOKIE_NAME` gives the name of the cookie. It can be any series of characters excluding whitespace, "=" signs, and semicolons. `COOKIE_VALUE` gives the value of the cookie. It can be any length, but has the same restriction on whitespace and funny characters. `EXPIRATION_DATE` tells the browser when the cookie is to expire. It needs to be in the official Internet date format (Chapter 2). Fortunately this format is compatible with the string returned by JavaScript's *Date* routines.

APPENDIX B (p. 13)

To maintain the cart in its entirety, we'll need to save each selected product's catalog number, description, price, and the number of items chosen. Because of the restrictions on the characters that can be contained within a cookie, we'll turn the cart into a long string in which the various items are separated by vertical bars, like this:

```
|E3219:Curry+Comb:21.95|2||GG9321:Clippers:12.50|1|
```

The item descriptions, including catalog number, name, and price, are packed together with colons in exactly the same way they were in the original catalog HTML file. This is followed by the number of items of this type in the cart. In this example there are three items in the cart: two curry combs and one clippers. Because whitespace isn't allowed within the cookie, we replace the space in the name "Curry Comb" with a plus sign.

To finish the cookie, we have to give it a name and an expiration date. We arbitrarily pick the name @CART and an expiration date one hour in the future. A typical shopping cart cookie looks like this:

```
@CART=|GG9321:Clippers:12.50|1|; expires=Thu Jul 04 12:06:52  
EDT 1995
```

The code changes needed to implement this cookie mechanism are actually pretty simple (Figure 10-14). The main changes are two new methods added to the "cart" object: `toCookie()`, which turns the cart into a cookie, and `fromCookie()`, which restores the cart from a cookie.

`toCookie()` uses JavaScript's string functions to build up the cookie one component at a time. It starts the new cookie with the string "@CART." Next it loops through the contents of the cart, adding the catalog number, description, price, and quantity to the cookie using the format described before. Finally, it calculates an expiration date one hour in the future using JavaScript's *Date* functions, and tacks on an "expires=" section to the cookie.

`fromCookie()` reverses this process. It uses JavaScript's string functions to locate the vertical bars and split out the items into separate variables. For each item a new *catEntry* object is created and added to the shopping cart.

Because spaces are a problem for cookies, we escape and unescape spaces in cookies using the utility functions `escape_spaces()` and `unescape_spaces()`. These functions simply examine each character in a string and convert spaces into "+" marks and back again.

Lastly, we need to call `toCookie()` and `fromCookie()` at the appropriate times to save and restore the user's shopping cart. The most natural time to save the cookie is when we rebuild the document that displays the user's shopping cart. A one-line modification to the `show()` method makes this happen. Now, in addition to opening up the HTML document, we set its cookie with the line:

```
aDoc.cookie = this.toCookie();
```

APPENDIX B (p. 14)

```

100.0 // Turn spaces into + signs
100.1 function escape_spaces (theString) {
100.2     var newString = "";
100.3     for (var i=0; i<theString.length; i++) {
100.4         if (theString.charAt(i) == " ") {
100.5             newString += "+";
100.6         } else {
100.7             newString += theString.charAt(i);
100.8         }
100.9     }
100.10    return newString;
100.11 }
100.12
100.13 // Turn + signs into spaces
100.14 function unescape_spaces (theString) {
100.15     var newString = "";
100.16     for (var i=0; i<theString.length; i++) {
100.17         if (theString.charAt(i) == "+") {
100.18             newString += " ";
100.19         } else {
100.20             newString += theString.charAt(i);
100.21         }
100.22     }
100.23    return newString;
100.24 }
100.25
100.26 // Turn the list into a cookie for transient storage of 1 hour
100.27 function toCookie() {
100.28     var theCookie, today;
100.29     theCookie = "@CART=";
100.30     for (var catNo in this.cart) {
100.31
100.32         var description = escape_spaces(this.entries[catNo].description);
100.33         var price = this.entries[catNo].price;
100.34         var quantity = this.cart[catNo];
100.35
100.36         // separate the various items with vertical bars
100.37         theCookie += "|" + catNo + ":" + description + ":" + price +
            "|" + quantity + "|";
100.38     }
100.39     expires = new Date;
100.40     expires.setTime(expires.getTime() + 1000*60*60); // one hour shelf
        life
100.41     theCookie += "; expires=" + expires;
100.42     return theCookie;
100.43 }
100.44
100.45 // Initialize ourselves from the cookie, if any
100.46 function fromCookie() {
100.47     var start = document.cookie.indexOf("@CART=");
100.48     start += "@CART=".length;
100.49     while (start < document.cookie.length) {
100.50         var firstBar = document.cookie.indexOf("|", start);

```



```

100.51     var secondBar = document.cookie.indexOf("|",firstBar+1);
100.52     var thirdBar = document.cookie.indexOf("|",secondBar+1);
100.53
100.54     var itemDesc = document.cookie.substring(firstBar+1,secondBar);
100.55     var quantity = document.cookie.substring(secondBar+1,thirdBar);
100.56
100.57     itemDesc = unescape_spaces(itemDesc);
100.58     for (var i = 1; i <= quantity; i++) {
100.59         this.add(new catEntry(itemDesc));
100.60     }
100.61     start = thirdBar + 1;
100.62 }
100.63 }

...

128     function show(aDoc) {
129         aDoc.clear();
129.1     aDoc.cookie = this.toCookie();
131     aDoc.open("text/html");
132     aDoc.writeln("<HTML><HEAD><TITLE>Current Shopping Cart</TITLE>
        </HEAD><BODY>");
133     aDoc.writeln("<H1>Current Shopping Cart</H1>");
134     aDoc.writeln(this.list());
135     aDoc.writeln("</BODY></HTML>");
136     aDoc.close();
137 }

....

156     // Constructor for the cart object
157     function cart() {
158         this.cart=new array();
159         this.entries = new array();
160         this.add=add;
161         this.remove=remove;
162         this.list=list;
163         this.show=show;
164         this.order=order;
165         this.make_orderForm = make_orderForm;
165.1     this.toCookie=toCookie;
165.2     this.fromCookie=fromCookie;
166     return this;
167 }

...

181     // GLOBAL INITIALIZATION - CREATE A NEW CART OBJECT
182     theCart = new cart();
182.1     theCart.fromCookie();

...

```

FIGURE 10.14 Giving the Shopping Cart Memory with a Magic Cookie

APPENDIX B (p. 16)


```
+1);  
+1);  
ndBar);  
rdBar);
```

This calls the `toCookie()` method to create a cookie containing the current shopping cart and stores the result in the document's `cookie` field. This cookie is subsequently grabbed by the browser and ferreted away into its database of cookies.

We restore the shopping cart from the cookie just once at global initialization time:

```
theCart = new cart();  
theCart.fromCookie();
```

Immediately after creating a new, empty shopping cart, we invoke its `fromCookie()` method. If the browser has sent us a cookie named "@CART", `fromCookie()` will retrieve it and use it to restore the shopping cart to its previous status.

Improvements to the Shopping Cart

In order to make this shopping cart example useful in the real world, you'll have to flesh out the order form a bit. The order form should perform field validation, and should accept a credit card number or some form of "e-money" using a secure protocol such as SSL. When it's submitted, the credit card number should be validated (or at least checked for the right number of digits) and entered into the vendor's order entry system.

Other parts of the script could stand some improvement as well. Currently, the script won't correctly handle catalog items that have one or more of the ":", "|", ";", or "=" characters in their names. The ":" character is used by the `catEntry()` method to separate the three fields of catalog descriptions, while the others have special meanings to cookies. In order to handle arbitrary item names, the script should implement general `escape()` and `unescape()` functions that recognize these characters and replace them with something safe.

Finally, the table that displays the user's current shopping cart could stand some improvement. When the shopping cart contains several items it would be natural to turn each item's name into a link so that when the user clicks on the item's name, the page that describes the product is reloaded into the "catalog" frame. This way the user can review her purchases and add or remove items from her order quickly. It's straightforward to extend the shopping cart object so that it saves the URL of the page that describes each item. You'll need to modify `catEntry()` so that it adds the current contents of `parent.catalog.document.location` to each item's description, and change `show()` so that it turns each item's name into a link.

APPENDIX B (p. 17)

**This Page is Inserted by IFW Indexing and Scanning
Operations and is not part of the Official Record**

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☒ **BLACK BORDERS**
- ☐ **IMAGE CUT OFF AT TOP, BOTTOM OR SIDES**
- ☐ **FADED TEXT OR DRAWING**
- ☐ **BLURRED OR ILLEGIBLE TEXT OR DRAWING**
- ☐ **SKEWED/SLANTED IMAGES**
- ☐ **COLOR OR BLACK AND WHITE PHOTOGRAPHS**
- ☐ **GRAY SCALE DOCUMENTS**
- ☐ **LINES OR MARKS ON ORIGINAL DOCUMENT**
- ☐ **REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY**
- ☐ **OTHER:** _____

IMAGES ARE BEST AVAILABLE COPY.

As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.